RTES Demo 2004 L2/3 Components and Setting

Jim Kowalkowski, etc.

1 Introduction

The purpose of this document is to identify and describe the software components that should be present in the next RTES demonstration system. Included are many of the component relationships and a proposal for how they might be distributed within the system and how many might be present. The concentration here is on trigger level 2/3.

A separate, but similar, document will address the Level-1 aspects of the RTES demonstration system for 2004.

1.1 Rationale

Level-2/3 was chosen because the management-level experiment control and monitoring concepts, features and software components match that of Level-1. In other words, the software elements necessary to configure the physical and logical layout of an experiment are similar; the messaging necessary to operate an experiment is similar. Passing messages around the system is similar. Level-2/3 is more straightforward to understand and do initial testing of tool scaling and API testing. It is also a good way to explore further the use of GME at BTeV. We can directly apply much of what is learned to Level-1.

1.2 Definitions

Crossing: the intersection of two "clouds" of particles, moving in opposite directions in the accelerator. As these clouds move through each other, occasionally one particle from each cloud will collide, sometimes spectacularly. The crossing rate is a constant (396 ns), determined by the accelerator. For BTeV, the average number of collisions per crossing will be six. (The actual number of collisions is Poisson distributed.)

Event: a crossing with at least one collision. Strictly speaking, each collision in a crossing is a separate "event." However, it is common usage to employ the word "event" (singular) to refer collectively to all of the collisions (events), how ever many there may be, associated with a specific crossing.

Reconstruction: the inversion of a set of discrete measurements from the detector, into a complex (and in some cases continuous) model of the physics interaction that gave rise to the discrete measurements. By analogy, to convert a collection of footprints in the sand into a hypothetical scenario of two people, and their dog, walking along the beach.

ADC: analog to digital converter; for the sake of this document, "ADC count" is synonymous with "raw data" from the detector.

1.3 Overview

The first demonstration system emphasized cooperation to get a system running that contained many of the RTES technologies in a BTeV Level-1 trigger-like set-

ting. It had many of the correct concepts present in a simplified form with little attention given to overall design issues and implementation. The next demonstration system should go a step further by containing more functions of the trigger and better component interfaces.

2 Goals

Why are we doing this? How do we measure success? This section needs to be refined.

High and Medium goals will be addressed; Low/No only as time permits.

2.1 High Priority Goals

1. Demonstrate the ability of GME to accurately model components of the trigger and predict overall system behavior when:

A. Connections between components are context dependent (no universal protocol and API) but appear to the GME user as simple relationships (the context dependence is hidden from user view),
B. There is a large number of components,
C. The source of trigger configuration information (algorithm modules, mode and control constants) comes from an external source.
D. Message exchanges are not explicitly coded in a particular language in dialog boxes
E. Trigger component physical layout, logical connections, and processing policies are distinct and can be easily chosen and combined at run time.

Will attempt to demonstrate (at least) 5 types of GME designs:

a) run control state machines - generating Python code for local, mid-level, and top nodes. Not hierarchical modeling. Rather, separate models which will be used at various levels. (ABCD)

b) GUIs - at least 3 Matlab based GUIs, for Run Control (command), Monitoring (display), and Faults (to/from ARMORs, VLAs). Easy to demonstrate user tailoring of the views. Will avoid writing Matlab code to keep the approach non-specific. (ABD)

c) datatype descriptors - for defining messages and structures. Generates headers, and (de)marshalling code for Elvin. Provides integration across multiple domains: Matlab-Elvin, Run Control-Elvin, ARMOR-Elvin, VLA-Elvin. (D)

d) system integration modeling (SIML) - overall description of the system, to allow naming and address assignment. Does not "define" the hierarch so much as describe it (although, in a homogeneous processor-switch environment, "describe" and "define" almost mean the same thing). Generates addresses, Elvin router configurations. (ABCE(logical))

e) ARMOR elements - models the types of elements (custom, standard) in use, but does not "define" custom element (i.e. can not just draw a picture to generate code that becomes a custom element). Generates configuration data and TCL for loading elements into ARMORs. Supports (generates wrapper for in-line?) custom element code. (C)

2. Demonstrate the ability to detect process failure (particularly filter program failure) and capture the data for an event in which there was a problem so that it may be used in the nightly build / unit test suite. Further, demonstrate remote access to support diagnosis following a detected failure.

Will demonstrate by running the demo and killing (hanging) processes. As many processes as possible will be covered by execution ARMORs.

Will implement an Elvin logger to save messages; part of Monitor (or Fault) GUI.

Will attempt to capture bad events to local files. But will not coalesce the data. The working assumption will be that by writing data locally, it will be easy for Dcache to make things appear to be global/central. But for this demo, Dcache will not be implement. We will just assume that it (or something like it) will be there some day.

Elvin messaging will provide some level of remote access. In particular, each Elvin publisher should respect a "level control" that will determine whether any given message should or should not be published. Capability, policy. If the Fault GUI can send Elvin messages to individual Elvin publishers, specifying individual "levels", then the operator will be able to turn up the volume from any selected node or process. Dynamically controllable remote monitoring.

Custom ARMORs may implement a fault-management hierarchy. Custom ARMORs may control VLAs. Custom ARMORs should communicate (to VLAs, to GUIs, to RC, to other Custom ARMORs) via Elvin.

11. Demonstrate scalability, or at least indicate the expected behavior as the system is scaled to full size.

Can demonstrate by adding nodes in Harry's farm, and by (specific) "experiments" on Vampire (developed elsewhere, but run for numerical results on Vampire).

Test message control and channel limits. How much can we afford to send as Elvin messages. Experiment with hierarchy, to determine how many (layers of) Elvin routers are needed. Show cost, and how costs scale with system size. Show message traffic, and how that scales with system size.

Demonstrate consistency between physical system and NS2 simulations.

2.2 Medium Priority Goals

3. Demonstrate use of standard protocols and APIs between components.

   By consistent use of rtes_ abstractions. No higher level code should use TCP or Elvin “raw”.

5. Characterize ARMOR and Element performance.

   Studied by running the demo. Demonstrate 10% (or less) resource consumed by RTES infrastructure

6. Demonstrate manual and automated removal of a worker for various hardware failures.

   Manual control via Fault GUI, etc. Kill process, hang process (requires processes to subscribe to "please hang yourself" Elvin messages), hang node (may require ARMOR daemon to subscribe to "please hang yourself").

   Automated behavior - ARMOR will try to restart crashed/killed processes. ARMORs will migrate in response to a failed node. Hot spare nodes to be identified by SIML/GME picture; list provided to FTM (ARMOR).

   Will not try to reboot hung machines.

9. Present precise APIs and encourage correct coding.

   Steve will (continue to) make this happen. At least correct coding…

10. Show adoption of a coherent build system that allows for library and system releases and nightly unit and integrated testing.

    By using CVS and makefiles for GME (Windows side); UDM (Windows/Linux side).

    Anyone should be able to build the runtime demo system on any appropriately prepared Linux platform: VU nodes, Harry's farm, or equivalent. Note that "prepared" may involve a nontrivial amount of 3rd party installation; this "preparation" will not be automated.

    Anyone should be able to build/edit/generate-python-or-matlab GME diagrams on any appropriately prepared Windows platform: Mike Haney's laptop, other. Note that "prepared" may involve a nontrivial amount of 3rd party installation; this "preparation" will not be automated.

2.3 Low Priority Goals

7. Demonstrate workers migrating from a partition from another as a run progresses and processor utilization drops. (Alternatively, demonstrate changing work-load assignments, controlled by trigger-need.)

> Not with respect to partitions or load balancing. Simulations (e.g. NS2) will study load balancing and task prioritization. No migration.

2.4 No Priority Goals

The following will probably not be done

4. Demonstrate dynamic construction of an experimental run (filter configuration and error handling policy choice).

> Without a "real" L2 filter, and physicist support, this goal is not practical. We might demonstrate how Run Control can pass information (configuration) to the filter app. And we will have startup scripts for the execution ARMORs to follow. But these scripts will probably not be generated by drawing pictures in GME.

8. Demonstrate the ability to record run activities (during an experiment) in a database for later playback postmortem analysis.

> As (part of) a mitigation, effort will be made to capture bad events and place them into a file. Playback, however, is not part of the plan. That would require a "real" L2 filter to be meaningful. Playback will be implemented when we revisit the ITCH and Source and provide real implementations for them.

3 Level 2/3 Diagrams

3.1 Inside the Filter Program

A filter program receives events (crossing results) one at a time. The event contains data from a single crossing from within the detector. These data are composed of blocks – one block from each of the subdetectors that are active in the system. The data can be thought of as ADC counts for detector cells that contain reading within a configured range. The event passes through a sequence of reconstruction algorithms (e.g. turn ADC counts into energy values and particle trajectories), followed by a decision sequence that is used to determine if this event has enough interesting physics in it to keep.

The diagram below shows some of the important components that will be present in a Level-2/3 program. A software framework allows these components to be plugged in and work together. The solid arrows represent real dataflow; the dashed arrows show logical flow of control and only appear in the trigger algo-

rithm path. A service is a component that can be used by any other component in the system whenever it wants. An event passes through the algorithm components; there may be several . Modules perform a function similar to algorithm components on events. BTeV currently has a simple prototype software framework and event data library.

The event path: An event enters via the event stream input module, which unloads or transforms portions of the data into structures used by algorithm components. The event is a complex database-like data structure. Pieces can be added to the event by name and pieces can be queried. The event is pushed through the algorithm components in the trigger algorithm path until a decision is made to accept or reject it. The detector information is highly packed; an algorithm will first uncompress the data. A rejection can happen early on, removing the need for all the algorithms to run. A final decision is made looking across the individual trigger paths (bit-wise OR for example). Events that pass are compressed or packed at the output module and written to output files depending on what trigger paths accepted the event. Periodically event files are sent to permanent storage. The event path shown in the diagram shows an extremely trivial trigger path. BTeV currently has a working prototype of the L2 pixel algorithm.

Data quality monitoring: As events flow through the filter program, the trigger algorithms keep statistics. The monitor data service is the keeper of these data structures and memory. At regular intervals, announcements are made through the control unit that statistics should be written out. BTeV currently has a prototype monitoring services based on the ROOT and HBOOK packages. [Question: are these centralized or distributed services?]

Problems: If problems or unique situations are found in the data or logic errors are detected by the trigger algorithm components, the components write information to the error logger service. The error logger service can synchronously deliver the message to another program [I suppose ARMOR-s can listen to these error messages?]. In general, trigger algorithm components do not determine themselves if things they discover are actually errors – it is up to the configuration of the control unit or the program that is hooked up to the error logger to determine if the situation is really an error. BTeV uses the Fermilab/CD developed ErrorLogger package.

Configuration and control: The configuration service organizes complex configuration information into data structures that are used by each component in the program, including the control unit. Each of these components will most likely have a large set of unique parameters. As the system runs, this service may be notified of system state changes, such as run segment change, stop of run, pause of run, start of run, calibration set ready, or increase debug level. BTeV currently only supports initial configuration using the Fermilab/CD developed package called RCP.

Other Services: The calibration/geometry services will most likely be skipped for this project. All the algorithms will need calibration constants that describe how to turn ADC counts into correct and meaningful physical quantities (characteristics

change with weather and beam exposure). They also need to know precisely where parts of the detector are (they move, sag, and change over time).

3.1.1 Required R&D

Incomplete.

1. "real" or dummy L2 and L3 algorithm code needs to be secured from BTeV sources.

2. Control unit. This element coordinates the execution of the trigger filter algorithm components. Keep in mind, that while the discussion above, and figure below, focuses on only the "L2" and "L3" components, the control unit must accommodate any number of conditioned-sequential, as well as unconditional components. This means both that there may be N (not necessarily 2) components, and the execution of component "i" may depend on "i-1", or may not. An interface to the ControlUnit must be defined that allows dynamic insertion of algorithm components in the control path.

3. the nature of the configuration data needed by the L2 and L3 algorithms needs to be characterized, and saved in a file or database. A formal mechanism for exposing the L2 and L3 configuration "hooks" must be developed. The configuration service must then be developed in a manner that allows changes in L2/L3 configuration needs to be satisfied **exclusively** through changes in the algorithm code, and in the configuration file/database. If done correctly (e.g. data-driven), the configuration service itself should never require recoding.

4. placeholders for the other services (geometry, calibration, file writing) should be created. While the body of code in these placeholder may be minimal, the interfaces should be fully developed.

5. the monitor data service and event logger service may be profoundly similar; careful consideration of class definitions should be made to maximize the code (class) sharing between these two. Need to carefully consider the distribution aspect of these services. A concerning aspect is how the service data from individual worker nodes is collated together, and the fault-tolerance properties of the “merge” node of the service data. The FT infrastructure may need to insert collation routines in the services based on the fault conditions, or even may need to have read interface to service data in order to make assessment regarding potential fault conditions.

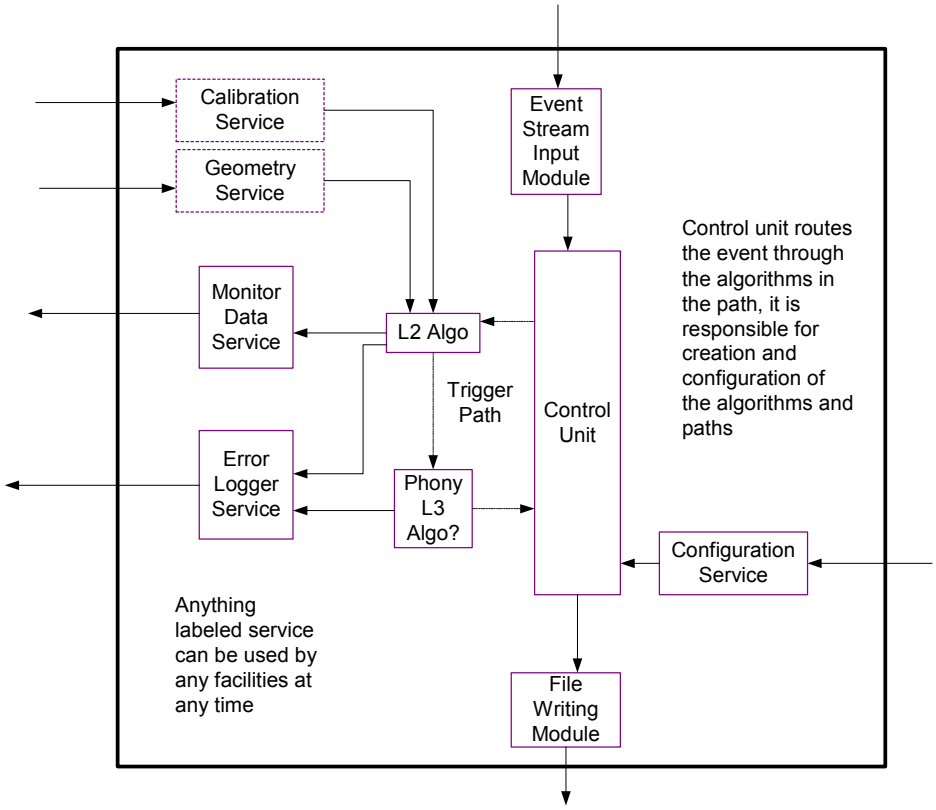

Figure 1: Inside the Filter Program

3.2 Inside a Worker

A worker node will most likely run one filter program instance per CPU. The worker also contains processes to move data and configuration on and off the node, and processes to start and stop programs and detect and respond to errors. Passed events and statistical information flow into temporary files on the node.

Event Builder: Event data will arrive at a single point (called an event builder – a separate executable) within the node. If data arrives in fragments, the event builder will coalesce the fragments into a whole event before dispatching it to one of the filter instances. The event builder will maintain a short queue of events, enough to keep the filter programs busy as much as possible. The event builder knows the protocol for requesting and receiving events. When the queue falls below an acceptable level, the builder requests one or more events to be transferred from the Level-1 buffer to this node.

Configuration/Data Movement: This is a caching/database tool used to efficiently transfer things such as executable versions, configuration sets, and calibration data from the central repositories for quick access during data taking. The information appears as a large virtual file system. Other processes on the node trigger the transfers. This subsystem may also be used to migrate data and statistics off the processor to a permanent store.

Run Control: This program manages the state of the node. It starts and stops programs depending on the state on the trigger. It receives instructions about what versions of executables to run and what configuration sets they require.

Filter Program: This was described in the previous section. It is important to recognize that a typical worker node will have 2 CPUs and hence will have two instances of the filter program (one per CPU), as shown in the figure below.

Fault Handling: This part of the system includes processes that watch other running programs and periodically report node health to other supervisor nodes, and processes that actively watch or monitor conditions in the machine or running application to make sure everything is operating smoothly. This subsystem will need to get system performance and health information from lower-level components.

3.2.1 Technology Choices

Data Movement: It is possible to use the product dcache or fake it with NFS. The advantage of dcache is that it provides a pseudo-filesystem where parts of it can be faulted in from unrelated technology (e.g. RDBMS or tape robot).

Statistics Cache: Most likely the information contained here will be in ROOT format (Histograms and Ntuples).

Filter Program Configuration: The RCP library (Fermilab Computing Division product).

System Monitoring: *Ganglia* is used by quite a few of the farms at Fermilab for recording things like process CPU/memory usage, network usage, and disk usage. PAPI provides an API to access the Intel/AMD on-chip high performance event counters. *Oprofile* can be used to record performance information about particular processes. *Linux-trace* is a Fermilab product that uses high performance kernel ring buffers to record kernel activities. It has logic analyzer-like triggers that allows snapshots of activity to be recorded based on programmed events. LM sensors provides an API to access the CPU/case temperature and fan speeds.

3.2.2 Protocols and IPC

Event Builder to Filter Program: The typical complete event could be as large at 250KB. Event buffer management should add as little overhead as possible to event processing. One method to accomplish this is to create a shared memory area between the Event Builder and the filter programs and then use messaging to coordinate access to areas of this shared memory. In other words, the communications here will be something efficient and custom for this application. The protocol must allow reconnection without restarting all players if a filter program crashes. A combination of Unix pipes, memory mapped files, and semaphores may be appropriate.

The following example implementation is offered not to define the solution, but only to illuminate the requirements. This example solution may or may not be optimal.

Consider a collection of 10 memory mapped files, each representing one "slot", and suitable for holding one event. The Event Builder is the only application which can write into these files; each of the filter program instances can only read. The Event Builder maintains a simple scoreboard to track the status of the slots; the scoreboard is also a memory mapped file.

The Event Builder pulls data from the L1 buffers, and arranges the various fragments (BLOBs) into a slot. When the entire event has been assembled, the Event Builder marks the slot as "ready" on the scoreboard, and if there are "too many" (configurable limit) other "empty" slots, the Event Builder proceeds to fill the next.

FilterProgram_1 completes processing, and messages the Event Builder to indicate completion and availability for a next assignment. The Event Builder knows (from the scoreboard) which slot had been assigned to FilterProgram_1, and marks this slot as "empty". The Event Builder then selects a "ready" slot, changes the scoreboard to read "assigned to Filter_1", and sends a reply message to FilterProgram_1 indicating which slot has been assigned to it. FilterProgram_1 then draws data from that slot, as it wishes. Upon completion, the cycle repeats.

If FilterProgram_1 crashes and restarts, it will send a message to the Event Builder asking for a **first** assignment (as opposed to the completion of a previous). The Event Builder recognizes, from the scoreboard and the **first** request, that FilterProgram_1 must have crashed. The Event Builder marks the slot as "**re**assigned to Filter_1" and sends the reply to the filter program. If the filter program crashes again (detected as a **first** request and an already **re**assigned slot), the Event Builder declares the event to be pathological, and forwards it to the Fault Handler for possible inclusion in the collection of test cases for future use.

[Sandeep: This sounds like a specific fault-mitigation behavior, a specific way of dealing with crashes related to data, shouldn't this be handled in a more coordinated manner with FMs/ARMORs/VLAs] [Mike: yes, somewhat… The Fault Handler is the VLA/ARMOR domain. Forwarding the event to the Fault Handler is essentially the same as sending a message to the VLA/ARMOR (ok, I'm reaching here). But remember, this is just an example to promote discussion. Nevertheless, I added the word "possible" to take the edge off of the specific resulting action.]

If the Event Builder crashes and restarts, it will understand the state of the system from the scoreboard (memory mapped file). It may be the case that the scoreboard is not perfectly accurate, depending on when and how the Event Builder crashed (e.g. whether a "flush" succeeded to write from memory to disk). The Event Builder will act in good faith, based on the state of the scoreboard. This may result in some lost events. Note that it may be advisable to mark the scoreboard to indicate the slot "being filled" with data from the Level-1 buffers.

Following a crash, that slot will be forfeit, and that event can be recognized as lost.

Into the Event Builder: A simple, efficient messaging library would be appropriate. PVM may work if it has good fault handling characteristics, such as reconnection upon communication failure or the master event distributor capable of coping with a worker node leaving or coming back into the system. A really simple version of CORBA ORB may actually work also. There is a project the out of the LHC experiment for doing this sort of thing (I had forgotten the name, Dinker knows the product). I remember it being a multi-channel system.

Standard Message Interface for Controls: The remaining links could use a trigger-wide messaging system. This includes Error Logger to Fault Manager, filter program Control Unit to Run Control, Run Control to upper-level Run Control, and Fault Manager to Run Control. This message passing can be slower and allow for more flexibility, lower coupling, and multiple language support. The computing division supports a few libraries: D0 message passing library (C++), Merlin (Luciano). A solution based on XMLRPC or SOAP may work well also. (*There will be another document to discuss message passing in more detail*.)

Notification: Perhaps this can be handled with the protocol/IPC discussion earlier, however I would recommend adoption of a common/unified notification service, with the ability to multicast (within the same node). Reason being there may be events such as run completion, which multiple programs on the node would be interested in knowing about.

3.2.3 Required R&D

Incomplete.

1. dcache
2. ganglia
3. shared memory event store, and scoreboard
4. notification service, publish/subscribe

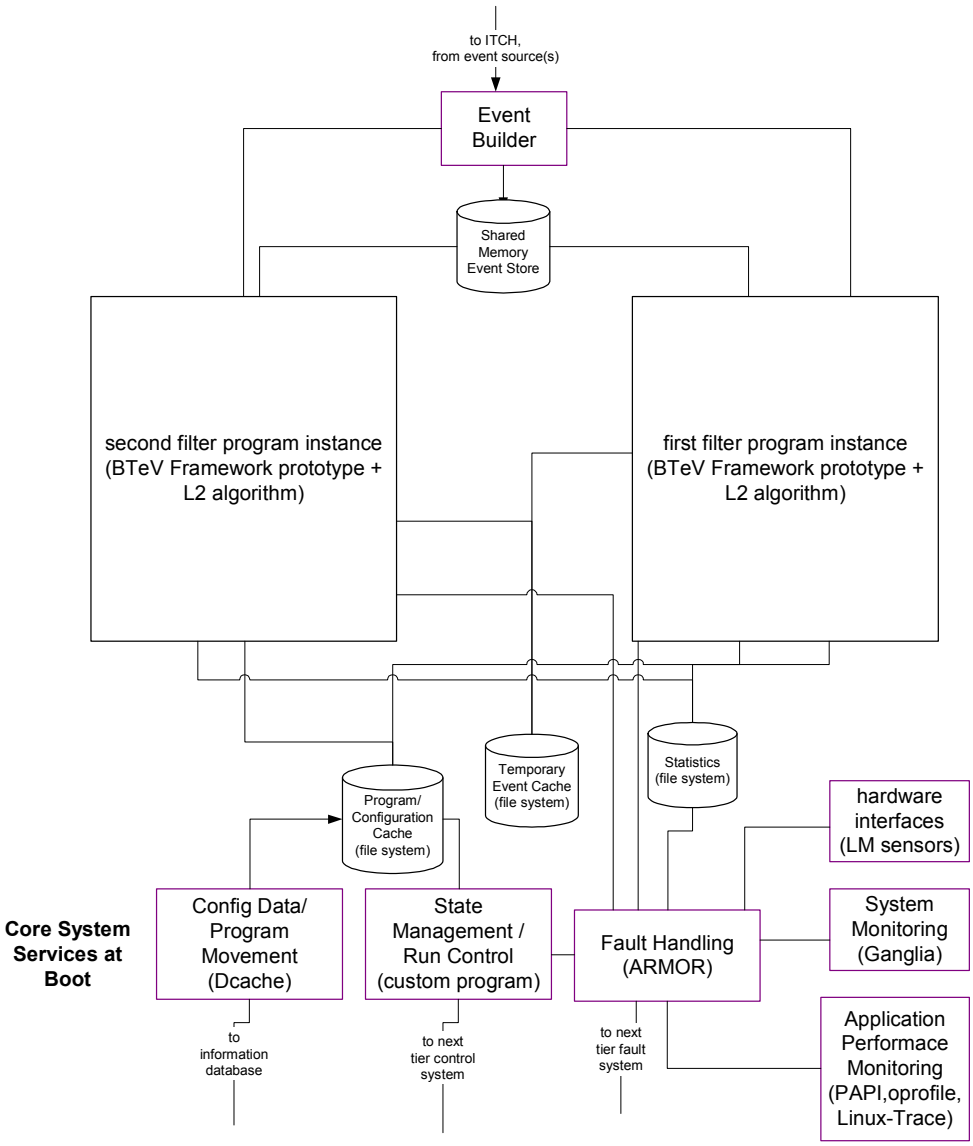

Figure 2: Inside an L2/L3 Worker Node (Processor)

3.3 A Proposed Setting

Multiple event sources mimic the trigger's Level-1 buffers. The real system will have one crossing (an entire event) spread over each of the L1 buffers (a fragment per BTeV sub-detector). We do not need to mimic this behavior on day one. We can move to it at a later time. Workers (i.e. the Event Builder in each worker) request events via the event distribution manager (ITCH). The distribution manager pulls data out of the buffers. This is somewhat difference than the current BTeV plan, which is for the ITCH to just make routing decision and let the buffer push data directly to worker nodes. Event data flows through the workers and out to the permanent store. In this demonstration, a single, large database contains all the information necessary to perform an experimental run – including executables, configuration, partition assignments, routing assignments, and fault

management geography data. Included is a hierarchy of controls, monitoring, and fault handling.

Event Source: Have a large set of events that will exercise various aspects of the system, including use of network bandwidth, CPU usage, and trip errors in the filter programs and utility programs in the main data path. The events can come from BTeVGEANT (BTeV detector simulation program) or be random data with a header describing the properties that it would have. The GEANT events can be made with qualities that will cause problems in the L2/3 filter algorithms.

ITCH: These machines connect the buffering machines in Level-1 to the nodes in Level-2/3, and perform the function of event distribution. Nodes in Level-2/3 will demand crossing data when they need more work. The ITCH will either tell the L1 buffer where to deliver data or collect the data and send it to the node. The latter may allow for more predictable resource use (network and memory on the buffer nodes).

L2/3 Processing Elements: These are the main workers running the filter programs described in previous sections. Depending on the system load and the test being run, an actual node can be configured to run one or more processing elements (workers).

Managers: A set of nodes, each in charge of a group of workers (processing elements). Functions such as administration, configuration commands, state notifications and management, and fault management will utilization these nodes. Watching for problems or trends across large areas of the trigger will require use of these nodes. We will define a hierarchical organization with 3-levels of hierarchy (Global, Managers, and Workers), the manager described here can be identified as a manager of a region of workers. In order to identify trends across a region, the manager will need access to region wide information: this includes monitor data/statistics, error log data, etc.

Global Trigger Manager: In the management hierarchy, this is the head node. This is the main gateway for control and management in the trigger. This node or set of nodes will know the protocols to talk with external systems.

Database Interface: This is the subsystem that manages access to information stored in databases, repositories, or external file systems. The databases contain information such as trigger configuration information, experimental run histories, node status data, and performance data. In this demonstration, we will need to simplify this. One way to simplify this is to remove as many DB queries as possible and provide information to workers and other subsystem in the file format it expects. The transport can probably start out as NFS, but later should move to a subsystem with caching behavior, such as dcache. This part of the system should not be used to inform systems of information that flows through the control system (e.g. state changes or announcement of what configuration to use). This system should not be used as a place to track things that change dynamically. The primary purpose, right now, is to allow programs on the workers to efficiently begin execution at the start of an experiment (a run) – without worrying about swamping the network of database component a run start. We want the

workers to stage or cache information that is frequently used across runs or information needed for the current run before the run actually starts. If NFS is used, then there will need to be a separate set of programs that populate the NFS area with programs and other important data from other sources like the CVS repository or release build area. If dcache is used, we may be able to install a set of routines that get triggered when cache faults occur. Dcache also manages the disk space you give it like a real cache. A real system may also allow more than just a file system abstraction – something like ad hoc SQL queries will probably be necessary also. This is something to discuss: should there be a database query method directly from the workers or should be controls/monitoring system message channels be used?

DAQ Element: The DAQ system will have many programs that can and will access the trigger for varying lengths of time. A run control daemon will most likely always be connected. A diagnostic program will connect for short periods of time. A monitoring program may connection periodically to take some performance measurements. The DAQ element represents a program that talks to the trigger directly. This involves authentication and known constraints on resource use.

Monitoring Interface: This is the GUI that control room personnel use to watch how well the system is performing and where they get notified of errors or problems.

Run Control Interface: This is the GUI that is used to configure and operate an experimental run.

Analysis: This is the set of tools used to do post mortem analysis of a run or compare performance data between two runs or to locate trends across runs.

3.3.1 Technology Choices

The management elements should use the same technology as the worker element state management program. Luciano (from the DAQ group) may have made some choices regarding the DAQ elements. We discussed using Matlab for all GUIs and analysis. We discussed starting with NFS to start out the database interface, knowing that this will be inadequate. The connection to the event sources should be the same as in the connection from the workers to the ITCH.

3.3.2 Protocols and IPC

The only new link introduced here is from global trigger manager to DAQ element. The default should be to use the same protocol as the management elements.

3.3.3 Required R&D

Incomplete.

1. Matlab for the RunControl and Monitoring interfaces

2. ITCH, Event source, event data set. Note that to get started, these can be combined into a single process (with a file containing a trivial number of events worth of data), and later elaborated into the multimode solution called for in the figure below.

3. database interface. Care should be exercised in allocating development effort to this task, as BTeV will almost certainly make choices which will not be identical with RTES. Therefore, the demo solution should be generic, and simple.

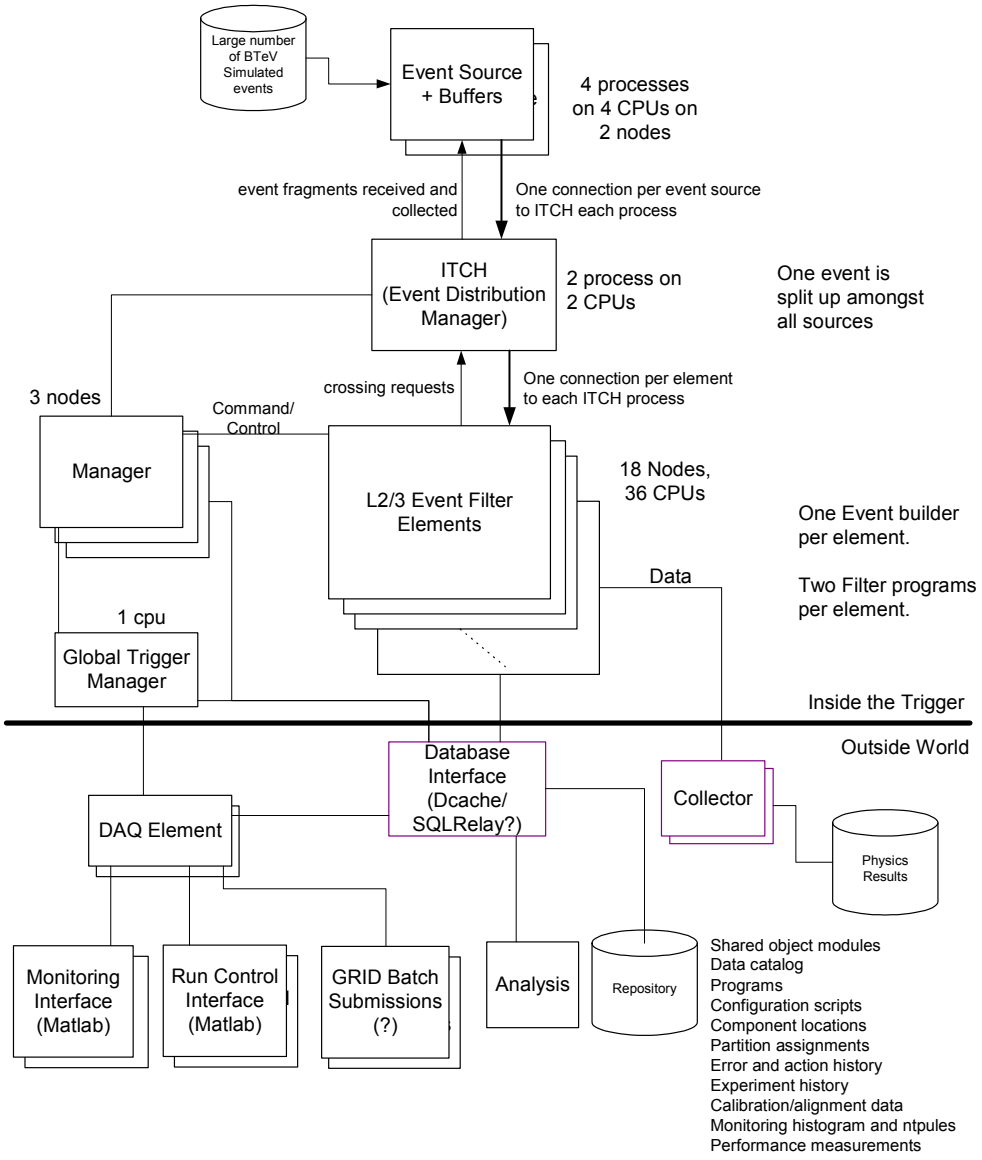

Figure 3: Overall System Architecture?

4 System states.

Incomplete.

5 Fault Models

The following fault models should be considered:

1. process crashes. Any process can crash; every process should be restartable. Execution ARMORs should be employed to cover almost everything… It would be desirable to be able to (remotely) poison any process to cause it to crash (as opposed to "ps -ax; kill -9"). Alternatively, a "grim reaper" script might be developed (to seek out by name, and terminate).

2. process loops. Especially in the filter algorithm(s), a(ny?) process can simply loop. This is not the same as a crash, although the recovery following detection may be equivalent.

3. bad connections. Event source to ITCH; ITCH to workers; workers to managers; workers to database; manager to DAQ/monitor/RunControl. It would be desirable to corrupt messages, and/or break channels in each of the above cases.

4. disk and database failures. Corruption, or simple loss of service.

5. worker node crash: an entire worker node crashes

6. manager node crash:

6 Relationship to a new Level-1 Project

Incomplete.

At this time, the L2/3 demonstration system is fully independent from any Level-1 project, past or present.

However, if resources are available, it should be relatively simple to allow Level-1 accept/reject messages to be sent to the ITCH, which in turn will simply alter the **apparent** event number of each event to track with the Level-1 activity. The actual data drawn from the event data set would be the same, regardless of Level-1, and every event from that set would be processed; the ITCH would not throw away data. However, through the expedient of creative labeling, the L2/3 system would "see" a discontinuous series of event numbers, corresponding to Level-1 accepts.

VLAs: there is one cross-over between Level-1, and L2/3 that merits discussion. It has been proposed that the Level-1 VLA solution find a way to capture and report the value of the program counter, and/or contents of the stack, in the event of a DSP worker crash. The monitoring interface at the top-most level will then decode this information (from a symbol table) to indicate, by name, what the worker was doing at the time of the crash. As the VLA concept is platform independent, it may be appropriate for the equivalent VLA solution (report-on-crash the node, process name, stack-trace, etc.) to be developed for the L2/3 demonstration system.

7 System Modeling

One of the key issues identified by the reviewers in the use of modeling tool, was a single model for the entire system, which makes it non-scalable, difficult for multi-user edits, and difficult to version different parts of the model. Another challenge arises from the fact that there is not a single unified component model that spans the entire system. A component model essentially defines what it means to be a “component”, what are its interfaces, and how components interact with each other. In the BTeV Trigger system, in general, and the L2/L3 demo system in particular there are many definitions of a component, and many different styles of interaction between components, for ex: an Event Builder, and a Filter Program, both are components which execute concurrently and interact through a specific protocol, which involves the asynchronous data transfers via shared memory, and notifications of processing completion; within a Filter Program, also there are several components which execute sequentially coordinated by a control unit, and exchange data in a pipelined fashion; the RunState manager is also another component that supervises/coordinates the execution of other components by sending start/stop notifications.

This makes it difficult to design a modeling language that addresses all the component models. Having a single grand-unified modeling language is clearly a non-scalable solution.

We propose a two-pronged approach to address this situation. We design different narrowly focused modeling languages for different component models, for ex: State-Machines for modeling mitigation behaviors, Dataflow modeling for the internals of the Filter Program, ARMOR element-interaction modeling for ARMORs. Individual components can be modeled using these modeling languages and these models are stored in separate models/files, which allows multiple users to independently developp/evolve/maintain/version these models. We also define an abstracted System Integration and Monitoring Language (SIML). This abstracted language shows the overall hierarchical organization and interaction, of the system components. Note that components will have a fairly loose definition in this language. The concrete models of these components, which could be GME models in the modeling languages listed earlier, or it could be just source code, are linked in to the abstract system model, through file system paths, and model object ids (or code references e.g. class name, function name, …). We use file system paths relative to the CVS module directory, for links. We can also provide GME plug-ins for creating links, and even displaying linked models/code. The SIML also allows capturing the debug/monitor information of a component, for ex: a State-Machine component would have a ‘CurrentState’, ‘TimeInCurrentState’, ‘StateHistory’, type of monitor elements, a Hardware (CPU) component would have a ‘CurrentTemp’, ‘CurrentUtilization’ etc. type of monitor elements. The idea is that during system operation a user should be able to drill into the system hierarchy, and ask for status update on the component. A GME plug-in queries the MonitoringData service for this information using the ID of the component, and animates it on the models. The model updates could be on-demand, or periodic refresh.

From a versioning and maintenance perspective the idea of using file/model links is a potentially viable one. Take for example the LFM behavior from the SC2003 demo. This and all other behaviors were modeled in a single model file which was in a different location from the component directory in the CVS. With the idea proposed above we can create one model for each behavior, store it as an XML file in the appropriate component directory. The interpreter (command-line form of interpretation is available) invocation could be integrated with the Build process, such that during the build the model is interpreted, its code generated, and the code is compiled.

8 Conclusion

Summarize the important points.